Chapter SEVENTEEN
Peeking, Mapping, Reducing and Collecting

Exam Objectives

Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method.
Save results to a collection using the collect method and group/partition data using the Collectors class.
Use of merge() and flatMap() methods of the Stream API.

peek()

`peek()` is a simple method:

```
Stream<T> peek(Consumer<? super T> action)
```

It just executes the provided `Consumer` and returns a new stream with the same elements of the original one.

Most of the time, this method is used with `System.out.println()` for debugging purposes (to see what's on the stream):

```
System.out.format("\n%d",
    IntStream.of(1, 2, 3, 4, 5, 6)
        .limit(3)
        .peek( i ->
            System.out.format("%d ", i) )
        .sum() );
```

The output:

```
1 2 3
6
```

Notice `peek()` is an intermediate operation. In the example, we can't use something like `forEach()` to print the values returned by `limit()` because `forEach()` is a terminal operation (and we couldn't call `sum()` anymore).

It's important to emphasize that `peek()` is intended to see the elements of a stream in a particular point of the pipeline, it's considered bad practice to change the stream in any way. If you want to do that, use the following method.

map()

`map()` is used to transform the value or the type of the elements of a stream:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)

IntStream mapToInt(ToIntFunction<? super T> mapper)

LongStream mapToLong(ToLongFunction<? super T> mapper)

DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

As you can see, `map()` takes a `Function` to convert the elements of a stream of type `T` to type `R`, returning a stream of that type `R`:

```
Stream.of('a', 'b', 'c', 'd', 'e')
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

The output:

```
97 98 99 100 101
```

There are versions for transforming to primitive types, for example:

```
IntStream.of(100, 110, 120, 130 ,140)
    .mapToDouble(i -> i/3.0)
    .forEach(i -> System.out.format("%.2f ", i));
```

Will output:

```
33.33 36.67 40.00 43.33 46.67
```

flatMap()

`flatMap()` is used to "flatten" (or combine) the elements of a stream into one (new) stream:

```
<R> Stream<R> flatMap(Function<? super T,
                      ? extends Stream<? extends R>> mapper)

DoubleStream flatMapToDouble(Function<? super T,
                      ? extends DoubleStream> mapper)

IntStream flatMapToInt(Function<? super T,
                      ? extends IntStream> mapper)

LongStream flatMapToLong(Function<? super T,
                      ? extends LongStream> mapper)
```

From its signature (and their primitive versions' signature) we can see that in contrast to `map()` (that returns a single value), `flatMap()` must return a `Stream`. If `flatMap()` returns `null`, an empty stream is returned instead.

Let's see how this work. If we have a stream of lists of characters:

```
List<Character> aToD = Arrays.asList('a', 'b', 'c', 'd');
List<Character> eToG = Arrays.asList('e', 'f', 'g');
Stream<List<Character>> stream = Stream.of(aToD, eToG);
```

And we want to convert all the characters to their `int` representation, we can't use `map()` anymore:

```
stream .map(c -> (int)c)
```

Because (as each element of the stream is passed to map) `c` represents an object of type `List<Character>`, not `Character`.

What we need to do is to get the elements of the lists into one stream and then convert each character to an `int`. Fortunately, the "combining" part is exactly what `flatMap()` does:

```
stream
    .flatMap(l -> l.stream())
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

So this code can output:

```
97 98 99 100 101 102 103
```

Using `peek()` after `flatMap()` may clarify how the elements are processed:

```
stream
    .flatMap(l -> l.stream())
    .peek(System.out::print)
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

As you can see from the output, the stream returned from `flatMap()` is passed through the pipeline, as if we were working with a stream of single elements and not with a stream of lists of elements:

```
a97 b98 c99 d100 e101 f102 g103
```

This way, with `flatMap()` you can convert a `Stream<List<Object>>` to `Stream<Object>`. However, the important concept is that this method returns a stream and not a single element (as `map()` does).

Reduction

A reduction is an operation that takes many elements and combines them (o *reduce* them) into a single value or object, and it is done by applying an operation multiple times.

Some examples of reductions are summing `N` elements, finding the maximum element of `N` numbers, or counting elements.

Like in the following example, where using a `for` loop, we reduce an array of numbers to their sum:

```
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

Of course, making reductions with streams instead of loops has more benefits, like easier parallelization and improved readability.

The Stream interface has two methods for reduction:

- `reduce()`
- `collect()`

We can implement reductions with both methods, but `collect()` help us to implement a type of reduction called *mutable reduction*, where a container (like a `Collection`) is used to accumulate the result of the operation.

reduce()

This method has three versions:

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity,
         BinaryOperator<T> accumulator)

<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Remember that a `BinaryOperator<T>` is equivalent to a `BiFunction<T, T, T>` , where the two arguments and the return type are all of the same types.

Let's start with the version that takes one argument. This is equivalent to:

```
boolean elementsFound = false;
T result = null;
for (T element : stream) {
    if (!elementsFound) {
        elementsFound = true;
        result = element;
    } else {
        result = accumulator.apply(result, element);
    }
return elementsFound ? Optional.of(result)
                     : Optional.empty();
```

This code just applies a function for each element, accumulating the result and returning an `Optional` wrapping that result, or an empty `Optional` if there were no elements.

Let's see a concrete example. We just see how a sum is a reduce operation:

```
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

Here, the accumulator operation is:

```
sum += n;
```

Or:

```
sum = sum + n;
```

Which translate to:

```
OptionalInt total = IntStream.of(1, 2, 3, 4, 5, 6)
                      .reduce( (sum, n) -> sum + n );
```

(Notice how the primitive version of stream uses the primitive version of `Optional`).

This is what happens step by step:

1. An internal variable that accumulates the result is set to the first element of a stream (`1`).
2. This accumulator and the second element of the stream (`2`) are passed as arguments to the `BinaryOperator` represented by the lambda expression `(sum, n) -> sum + x`.
3. The result (`3`) is assigned to the accumulator.
4. The accumulator (`3`) and the third element of the stream (`3`) are passed as arguments to the `BinaryOperator`.
5. The result (`6`) is assigned to the accumulator.
6. Steps 4 and 5 are repeated for the next elements of the stream until there are no more elements.

However, what if you need to have an initial value? For cases like that, we have the version that takes two arguments:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

The first argument is the initial value, and it is called identity because strictly speaking, this value must be an identity for the accumulator function, in other words, for each value `v`, `accumulator.apply(identity, v)` must be equal to `v`.

This version of `reduce()` is equivalent to:

```
T result = identity;
for (T element : stream)  {
    result = accumulator.apply(result, element);
}
return result;
```

Notice that this version does not return an `Optional` object because if the stream empty, the identity value is returned.

For example, the sum example can be rewritten as:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
               .reduce( 0,
                       (sum, n) -> sum + n ); // 21
```

Or using a different initial value:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
                .reduce( 4,
                        (sum, n) -> sum + n ); // 25
```

However, notice that in the last example, the first value cannot be considered an identity (as in the first example) since, for instance, `4 + 1` is not equal to `4`.

This can bring some problems when working with parallel streams, which we'll review in the next chapter.

Now, notice that with these versions, you take elements of type `T` and return a reduced value of type `T` also.

However, if you want to return a reduced value of a different type, you have to use the three arguments version of `reduce()`:

```
<U> U reduce(U identity,
             BiFunction<U,? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

(Notice the use of types `T` and `U`)

This version is equivalent to:

```
U result = identity;
for (T element : stream) {
    result = accumulator.apply(result, element)
}
return result;
```

Consider for example that we want to get the sum of the length of all strings of a stream, so we are getting strings (type `T`), and we want an integer result (type `U`).

In that case, we use `reduce()` like this:

```
int length =
    Stream.of("Parallel", "streams", "are", "great")
        .reduce(0,
                (accumInt, str) ->
                   accumInt + str.length(), //accumulator
                (accumInt1, accumInt2) ->
                   accumInt1 + accumInt2);//combiner
```

We can make it clearer by adding the argument types:

```
int length =
  Stream.of("Parallel", "streams", "are", "great")
      .reduce(0,
            (Integer accumInt, String str) ->
               accumInt + str.length(), //accumulator
            (Integer accumInt1, Integer accumInt2) ->
               accumInt1 + accumInt2);//combiner
```

As the accumulator function adds a mapping (transformation) step to the accumulator function, this version of the `reduce()` can be written as a combination of `map()` and the other versions of the `reduce()` method (you may know this as the *map-reduce* pattern):

```
int length =
    Stream.of("Parallel", "streams", "are", "great")
        .mapToInt(s -> s.length())
```

```
        .reduce(0,
                (sum, strLength) ->
                    sum + strLength);
```

Or simply:

```
int length = Stream.of("Parallel", "streams", "are", "great")
                .mapToInt(s -> s.length())
                .sum();
```

Because, in fact, the calculation operations that we learned about in the last chapter are implemented as reduce operations under the hood:

- `average`
- `count`
- `max`
- `min`
- `sum`

Also, notice that if return a value of the same type, the combiner function is no longer necessary (it turns out that this function is the same as the accumulator function) so, in this case, it's better to use the two argument version.

It's recommended to use the three version `reduce()` method when:

- Working with parallel streams (more of this in the next chapter)
- Having one function as a mapper and accumulator is more efficient than having separate mapping and reduction functions.

collect()

This method has two versions:

```
<R,A> R collect(Collector<? super T,A,R> collector)

<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

The first version uses predefined collectors from the `Collectors` class while the second one allows you to create your own collectors. Primitive streams (like `IntStream`), only have this last version of `collect()`.

Remember that `collect()` performs a mutable reduction on the elements of a stream, which means that it uses a mutable object for accumulating, like a `Collection` or a `StringBuilder`. In contrast, `reduce()` combines two elements to produce a new one and represents an immutable reduction.

However, let's start with the version that takes three arguments, as it's similar to the `reduce()` version that also takes three arguments.

As you can see from its signature, first, it takes a `Supplier` that returns the object that will be used as a container (accumulator) and returned at the end.

The second parameter is an accumulator function, which takes the container and the element to be added to it.

The third parameter is the combiner function, which merges the intermediate results into the final one (useful when working with parallel streams).

This version of `collect()` is equivalent to:

```
R result = supplier.get();
for (T element : stream) {
    accumulator.accept(result, element);
}
return result;
```

For example, if we want to "reduce" or "collect" all the elements of a stream into a `List`, we can do it this way:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           () -> new ArrayList<>(),// Creating the container
           (l, i) -> l.add(i), // Adding an element
           (l1, l2) -> l1.addAll(l2) // Combining elements
        );
```

We can make it clearer by adding the argument types:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           () -> new ArrayList<>(),
           (List<Integer> l, Integer i) -> l.add(i),
           (List<Integer> l1, List<Integer> l2) -> l1.addAll(l2)
        );
```

Or we can also use method references:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           ArrayList::new,
           ArrayList::add,
           ArrayList::addAll
        );
```

Collectors

The previous version of `collect()` is useful to learn how collectors work, but in practice, it's better to use the other version.

Some common collectors of the `Collectors` class are:

Method	Returned value from collect()	Description
`toList`	`List`	Accumulates elements into a `List`.
`toSet`	`Set`	Accumulates elements into a `Set`.
`toCollection`	`Collection`	Accumulates elements into a `Collection` implementation.
`toMap`	`Map`	Accumulates elements into a `Map`.
`joining`	`String`	Concatenates elements into a `String`.

`groupingBy`	`Map<K, List<T>>`	Groups elements of type `T` in lists according to a classification function, into a map with keys of type `K`.
`partitioningBy`	`Map<Boolean, List<T>>`	Partitions elements of type `T` in lists according to a predicate, into a map.

Since calculation methods can be implementing as reductions, the `Collectors` class also provides them as collectors:

Method	Returned value from collect()	Description

`averagingInt`
`averagingLong`
`averagingDouble`
	`Double`	Returns the average of the input elements.

`counting`	`Long`	Counts the elements of input elements.
`maxBy`	`Optional<T>`	Returns the maximum element according to a given `Comparator`.
`minBy`	`Optional<T>`	Returns the minimum element according to a given `Comparator`.

`summingInt`
`summingLong`
`summingDouble`
	`Integer`
	`Long`
	`Double`
		Returns the sum of the input elements.

This way, we can rewrite our previous example:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           ArrayList::new,
           ArrayList::add,
           ArrayList::addAll
    );
```

As:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(Collectors.toList()); // [1, 2, 3, 4, 5]
```

Since all these methods are static, we can use `static` imports:

```
import static java.util.stream.Collectors.*;
...
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(toList()); // [1, 2, 3, 4, 5]
```

If we want to collect the elements into a `Set`:

```
Set<Integer> set =
    Stream.of(1, 1, 2, 2, 2)
        .collect(toSet()); // [1, 2]
```

If we want to create another `Collection` implementation:

```
Deque<Integer> deque =
    Stream.of(1, 2, 3)
        .collect(
            toCollection(ArrayDeque::new)
    ); // [1, 2, 3,]
```

If we are working with streams of `String` s, we can join all the elements into one `String` with:

```
String s = Stream.of("a", "simple", "string")
              .collect(joining()); // "asimplestring"
```

We can also pass a separator:

```
String s = Stream.of("a", "simple", "string")
             .collect(joining(" ")); // " a simple string"
```

And a prefix and a suffix:

```
String s = Stream.of("a", "simple", "string")
             .collect(
                 joining(" ", "This is ", ".")
             ); // "This is a simple string."
```

In the case of maps, things get a little more complicated, because depending on our needs, we have three options.

In the first option, `toMap()` takes two arguments (I'm not showing the return types because they are hard to read and don't provide much value anyway):

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper)
```

Both `Functions` take an element of the stream as an argument and return the key or the value of an entry of the `Map` , for example:

```
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(
            toMap(i -> i, // Key
                  i -> i * 2 // Value
            )
        );
```

Here, we're using the element (like `1`) as the key, and the element multiplied by two as the value (like `2`).

We can also write `i -> i` as `Function.identity()` :

```
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(
            toMap(Function.identity(), // Key
                  i -> i * 2 // Value
            )
        );
```

`java.util.function.Function.identity()` returns a function that always returns its input argument, in other words, it's equivalent to `t -> t`.

But what happens if more than one element is mapped to the same key, like in:

```
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(i -> i % 2, // Key
                       i -> i // Value
                 )
        );
```

Java won't know what to do, so an exception will be thrown:

```
Exception in thread "main" java.lang.IllegalStateException: Duplicate key 1
   at java.util.stream.Collectors.lambda$throwingMerger$113(Collectors.java:133)
   at java.util.stream.Collectors$$Lambda$3/303563356.apply(Unknown Source)
   at java.util.HashMap.merge(HashMap.java:1245)
   at java.util.stream.Collectors.lambda$toMap$171(Collectors.java:1320)
```

For those cases, we use the version that takes three arguments:

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper,
      BinaryOperator<U> mergeFunction)
```

The third argument is a function that defines what to do when there's a duplicate key. For example, we can create a `List` to append values:

```
Map<Integer, List<Integer>> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(
                    i -> i % 2,
                    i -> new ArrayList<Integer>(Arrays.asList(i)),
                    (list1, list2) -> {
                       list1.addAll(list2);
                       return list1;
                    }
                 )
         );
```

This will return the following map:

```
{0=[2, 4, 6], 1=[1, 3, 5]}
```

The third version of `toMap()` takes all these arguments plus one that returns a new, empty `Map` into which the results will be inserted:

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper,
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapSupplier)
```

So we can change the default implementation (`HashMap`) to `ConcurrentHashMap` for example:

```
Map<Integer, List<Integer>> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(
                    i -> i % 2,
```

```
                    i -> new ArrayList<Integer>(Arrays.asList(i)),
                    (list1, list2) -> {
                        list1.addAll(list2);
                        return list1;
                    },
                    ConcurrentHashMap::new
                )
        );
```

About the calculation methods, they are easy to use. Except for `counting()`, they either take a `Function` to produce a value to apply the operation, or (in the case of `maxBy` and `minBy`) they take a `Comparator` to produce the result:

```
double avg = Stream.of(1, 2, 3)
                .collect(averagingInt(i -> i * 2)); // 4.0

long count = Stream.of(1, 2, 3)
                .collect(counting()); // 3

Stream.of(1, 2, 3)
    .collect(maxBy(Comparator.naturalOrder()))
    .ifPresent(System.out::println); // 3

Integer sum = Stream.of(1, 2, 3)
    .collect(summingInt(i -> i)); // 6
```

groupingBy()

The `Collectors` class provides two functions to group the elements of a stream into a list, in a kind of an `SQL GROUP BY` style.

The first method is `groupingBy()` and it has three versions. This is the first one:

```
groupingBy(Function<? super T,? extends K> classifier)
```

It takes a `Function` that classifies elements of type `T`, groups them into a list and returns the result in a `Map` where the keys (of type `K`) are the `Function` returned values.

For example, if we want to group a stream of numbers by the range they belong (tens, twenties, etc.), we can do it with something like this:

```
Map<Integer, List<Integer>> map =
   Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
      .collect( groupingBy (i -> i/10 * 10 ) );
```

The moment you compare this code with the traditional way to it (with a `for` loop), it's when you realize the power of streams:

```
List<Integer> stream =
    Arrays.asList(2,34,54,23,33,20,59,11,19,37);
Map<Integer, List<Integer>> map = new HashMap<>();

for(Integer i : stream) {
    int key = i/10 * 10;
    List<Integer> list = map.get(key);

    if(list == null) {
        list = new ArrayList<>();
        map.put(key, list);
    }
```

```
    list.add(i);
}
```

Either way, those will return the following map:

```
{0=[2], 50=[54,59], 20=[23,20], 10=[11,19], 30=[34,33,37]}
```

The second version takes a *downstream collector* as an additional argument:

```
groupingBy(Function<? super T,? extends K> classifier,
           Collector<? super T,A,D> downstream)
```

A d*ownstream collector* is a collector that is applied to the results of another collector.

We can use any collector here, for instance, to count the elements in each group of the previous example:

```
Map<Integer, Long> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
        .collect(
            groupingBy(i -> i/10 * 10,
                       counting()
            )
        );
```

(Notice how the type of the values of the `Map` change to reflect the type returned by the downstream collector, `counting()`)

This will return the following map:

```
{0=1, 50=2, 20=2, 10=2, 30=3}
```

We can even use another `groupingBy()` to classify the elements in a second level. For instance, instead of counting, we can further classify the elements in even or odd:

```
Map<Integer, Map<String, List<Integer>>> map =
   Stream.of(2,34,54,23,33,20,59,11,19,37)
       .collect(groupingBy(i -> i/10 * 10,
                  groupingBy(i ->
                              i%2 == 0 ? "EVEN" : "ODD")
                )
       );
```

This will return the following map (with a little formatting):

```
{
    0 = {EVEN=[2]},
   50 = {EVEN=[54], ODD=[59]},
   20 = {EVEN=[20], ODD=[23]},
   10 = {ODD=[11, 19]},
   30 = {EVEN=[34], ODD=[33, 37]}
}
```

The key of the high-level map is an Integer because the first `groupingBy()` returns an `Integer` .

The type of the values of the high-level map changed (again) to reflect the type returned by the downstream collector, `groupingBy()` .

In this case, a `String` is returned so this will be the type of the keys of the second-level map, and since we are working with a stream of Integers, the values have a type of `List<Integer>`.

Seeing the output of these examples, you may be wondering, is there a way to have the result ordered?

Well, `TreeMap` is the only implementation of `Map` that is ordered. Fortunately, the third version of `groupingBy()` add a `Supplier` argument that let us choose the type of the resulting `Map`:

```
groupingBy(Function<? super T,? extends K> classifier,
           Supplier<M> mapFactory,
           Collector<? super T,A,D> downstream)
```

This way, if we pass an instance of `TreeMap`:

```
Map<Integer, Map<String, List<Integer>>> map =
    Stream.of(2,34,54,23,33,20,59,11,19,37)
       .collect( groupingBy(i -> i/10 * 10,
                 TreeMap::new,
                 groupingBy(i -> i%2 == 0 ? "EVEN" : "ODD")
                )
       );
```

This will return the following map:

```
{
    0 = {EVEN=[2]},
   10 = {ODD=[11, 19]},
   20 = {EVEN=[20], ODD=[23]},
   30 = {EVEN=[34], ODD=[33, 37]},
   50 = {EVEN=[54], ODD=[59]}
}
```

partitioningBy()

The second method for grouping is `partitioningBy()`.

The difference with `groupingBy()` is that `partitioningBy()` will return a `Map` with a `Boolean` as the key type, which means there are only two groups, one for true and one for `false`.

There are two versions of this method. The first one is:

```
partitioningBy(Predicate<? super T> predicate)
```

It partitions the elements according to a `Predicate` and organizes them into a `Map<Boolean, List<T>>`.

For example, if we want to partition a stream of numbers by the ones that are less than `50` and the ones that don't, we can do it this way:

```
Map<Boolean, List<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31)
        .collect(partitioningBy(i -> i < 50));
```

This will return the following map:

```
{false=[65, 77, 89], true=[45, 9, 12, 31, 12]}
```

As you can see, because of the `Predicate`, the map will always have two elements.

And like `groupingBy()`, this method has a second version that takes a downstream collector.

For example, if we want to remove duplicates, we just have to collect the elements into a `Set` like this:

```
Map<Boolean, Set<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31, 12)
        .collect(
            partitioningBy(i -> i < 50,
                           toSet()
            )
        );
```

This will produce the following `Map`:

```
{false=[65, 89, 77], true=[9, 12, 45, 31]}
```

However, unlike `groupingBy()`, there's no version that allows us to change the type of the `Map` returned. But it doesn't matter, you only have two keys that you can get with:

```
Set<Integer> lessThan50 = map.get(true);
Set<Integer> moreThan50 = map.get(false);
```

Key Points

- `peek()` executes the provided `Consumer` and returns a new stream with the same elements of the original one. Most of the time, this method is used for debugging purposes.
- `map()` is used to transform the value or the type of the elements of a stream through a provided `Function`.
- `flatMap()` is used to "flatten" (or combine) the elements of a stream into one (new) stream. In contrast to `map()` (that returns a single value), `flatMap()` must return a `Stream`.
- A reduction is an operation that takes many elements and combines them (o reduce them) into a single value or object.
- `reduce()` performs a reduction on the elements of a stream using an accumulation function, an optional identity, and an also optional combiner function.
- `collect()` implements a type of reduction called *mutable reduction*, where a container (like a `Collection`) is used to accumulate the result of the operation.
- The `Collectors` class provides static methods such as `toList()` and `toMap()` to create a collection or a map from a stream and some calculation methods like `averagingInt()`.
- `Collectors.groupingBy()` groups the elements of a stream using a given `Function` as a classifier. It can also receive a *downstream collector* to create another level of classification.
- You can also group (or partition) the elements in a stream based on a condition (`Predicate`) using the `Collectors.partitioningBy()` method.

Self Test

1. Given:

```
public class Question_17_1 {
    public static void main(String[] args) {
        Map<Boolean, List<Integer>> map =
           Stream.of(1, 2, 3, 4)
              .collect(partitioningBy(i -> i < 5));
        System.out.println(map);
    }
}
```

What is the result?

A. `{true=[1,2, 3, 4]}`
B. `{false=[], true=[1, 2, 3, 4]}`
C. `{false=[1,2, 3, 4]}`
D. `{false=[1, 2, 3, 4], true=[]}`

2. Given:

```
groupingBy(i -> i%3, toList())
```

Which of the following is equivalent?

A. `partitioningBy(i -> i%3 == 0, toList())`
B. `partitioningBy(i -> i%3, toList())`
C. `groupingBy(i -> i%3 == 0)`
D. `groupingBy(i -> i%3)`

3. Given:

```
public class Question_17_3 {
    public static void main(String[] args) {
        Stream.of("aaaaa", "bbbb", "ccc")
           .map(s -> s.split(""))
           .limit(1)
           .forEach(System.out::print);
    }
}
```

What is the result?

A. `aaaaa`
B. `abc`
C. `a`
D. None of the above

4. Given:

```
public class Question_17_4 {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("a", "b", "c")
                .flatMap(s -> Stream.of(s, s , s))
                .collect(Collectors.toList())
        );
    }
}
```

What is the result?

A. `[a, a, a, b, b, b, c, c, c]`
B. `[a, a, a]`

C. `[a, b, c]`
D. Compilation fails

5. Which of the following is the right way to implement `OptionalInt min()` with a reduce operation?
A. `reduce((a,b) -> a > b)`
B. `reduce(Math::min)`
C. `reduce(Integer.MIN_VALUE, Math:min)`
D. `collect(Collectors.minBy())`

6. Which of the following is a correct overload of the `reduce()` method?
A.

```
T reduce(BinaryOperator<T> accumulator)
```

B.

```
Optional<T> reduce(T identity,
           BinaryOperator<T> accumulator)
```

C.

```
<U> U reduce(BinaryOperator<T> accumulator,
               BinaryOperator<U> combiner)
```

D.

```
<U> U reduce(U identity,
           BiFunction<U,? super T,U> accumulator,
           BinaryOperator<U> combiner)
```

7. Given:

```
public class Question_17_7 {
    public static void main(String[] args) {
        Map<Integer, Map<String, List<Integer>>> map =
            Stream.of(56, 54, 1, 31, 98, 98, 16)
               .collect(groupingBy(
                           i -> i%10,
                           TreeMap::new,
                           partitioningBy(i -> i > 5)
                       )
               );
        System.out.println(map);
    }
}
```

What is the result?
A.

```
{
   6={false=[], true=[56, 16]},
   4={false=[], true=[54]},
   1={false=[1], true=[31]},
   8={false=[], true=[98]}
}
```

B.

```
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]},
   6={false=[], true=[56, 16]},
```

```
   8={false=[], true=[98]}
}
```

C.

```
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]},
   6={false=[], true=[56, 16]},
   8={false=[], true=[98, 98]}
}
```

D.

```
{
   1={false=[1], true=[31]},
   4={false=[], true=[54]}
}
```

Open answers page

---

16. Stream Operations on Collections 18. Parallel Streams